An Observationally Complete Program Logic for Imperative Higher-Order Functions

Kohei Honda Nobuko Yoshida Martin Berger

Abstract

We propose a simple compositional program logic for an imperative extension of call-by-value PCF, built on Hoare logic and our preceding work on program logics for pure higher-order functions. A central feature of the logic is its systematic use of names and operations on them. This allows precise and general description of complex higher-order imperative behaviour in assertions. The proof rules of the logic exactly follow the syntax of the language and can cleanly embed, justify and extend the standard proof rules for total correctness of Hoare logic. The logic offers a foundation for general treatment of aliasing and local state on its basis, with minimal and clean extensions. After establishing soundness, we prove that valid assertions for programs completely characterise their behaviour up to observational congruence, which is established using a variant of finite canonical forms. The use of the logic is illustrated through reasoning examples which have been hard to assert and infer using existing program logics.

1. Introduction

Imperative higher-order functions, syntactically embodied by imperative extensions of the λ-calculus, have been one of the major topics in the study of semantics and types of programming languages for decades. They are a cornerstone of richly typed functional programming languages such as ML [36] and Haskell [3] and are central to the semantic analysis of procedural, object-oriented and even low-level languages [1, 15, 30, 37, 41, 46]. The significance of combining imperative features and higher-order functions lies in their distilled presentation of key elements of sequential program behaviour, amenable for theoretical inquiry. This analytical nature makes it possible to develop rigorous operational semantics for their dynamics [29, 36, 43], a rich class of type disciplines [36, 41] and powerful operational reasoning techniques [32, 42].

Given these achievements, a natural question is if we can carry out a similar development for logical methods for compositional reasoning in the tradition of Hoare logic [12, 19, 38]. In Hoare logic, assertions on programs offer a method for precisely describing properties of programs independent from the latter's textual details, with proof rules enabling verification of valid assertions following the syntactic structure of target programs. Hoare logic has however been mainly developed for first-order imperative programs: its extension to accommodate general higher-order procedures has been known to be a subtle problem [7, 11, 14, 34, 35].

The present paper introduces a simple compositional program logic for an imperative extension of call-by-value PCF, built on Hoare logic [19] and our preceding work on logics for pure higher-order functions [21, 25]. The assertions in the logic precisely describe behaviour of imperative higher-order procedures up to the observational equivalence, while proof rules enable compositional derivation of valid assertions. As far as we know, this is the first time a compositional program logic for imperative higher-order functions in full type hierarchy has been developed. The logical articulation of higher-order behaviour is rigorously stratified, starting from pure functions [21, 25] and treating each significant imperative element, including state change, aliasing and local state, with an incremental enrichment of the assertion language and proof rules. The logic enjoys clean semantic status in the sense that valid assertions for a program precisely characterise its observational behaviour up to the contextual congruence [18, 34].

A syntactically simple extension of the Floyd-Hoare tradition for treating higher-order behaviour is that assertions in our logic not only talk about first-order data stored in imperative variables, as in Hoare's logic and its standard extensions, but also about arbitrary higher-order imperative behaviours, which may be fed as arguments to procedures, denoted by functional variables and stored in imperative variables. Having programs' behaviour as part of the universe of discourse is essential for reasoning about practical programs since functionalities of a higher-order program often crucially depend on the combined behaviour of the programs it uses. As an example, consider $\texttt{twice} \stackrel{\text{def}}{=} \lambda f^{\alpha\Rightarrow\alpha}.\lambda x^{\alpha}.(f(fx))$, with α being a higher-order type. The behaviour of this program depends on the potentially complex interplay between f and x, all the more so if they have side effects.

Thus our logical language fully embraces higher-order behaviours and data structures as target of description, which is done by naming behaviours by variables and asserting on them, rather than having their textual representation (programs) in assertions.

Let us present three simple, but non-trivial programming examples, a clean and rigorous behavioural description by a logical means is set to be one of the challenges in our present inquiry (we use notations from standard textbooks [16, 41]).

$$\begin{array}{l}\texttt{closureFact} \stackrel{\text{def}}{=} \\ \quad \mu f^{\mathsf{Nat}\Rightarrow\mathsf{Unit}}.\lambda x^{\mathsf{Nat}}.\,\texttt{if}\ x=0 \\ \qquad\qquad\qquad\qquad \texttt{then}\ y:=\lambda().1 \\ \qquad\qquad\qquad\qquad \texttt{else}\ y:=\lambda().(f(x-1);x\times(!y)()\,)\end{array}$$

Above $()$ is the unique constant of type Unit and $\lambda().N$ denotes $\lambda z^{\mathsf{Unit}}.N$ with z fresh. When invoked as e.g. `closureFact` 3, the program stores a procedure in the imperative variable y. If we further invoke this stored procedure as $(!y)()$, then `closureFact` is called again with the argument $3-1=2$, after which a program stored in y *at that time* is invoked, so that the multiple of x and the value returned by that program is calculated and is given as the final return value. The intension is that this final value should be the factorial of 3. The observable behaviour of `closureFact` can be informally described as: *When the program is fed with a number n, it stores in y a closure which, when invoked with* $()$*, will return the factorial of n.* Note that inside the body of `closureFact`, a free variable f and the content of an imperative variable y are used non-trivially. In particular, the correctness of this program crucially depends on how y is updated sequentially in an orderly manner.

Next we consider another nonstandard, but terser, factorial program, using Landin's idea [29] to realise a recursion by circular references.

$$\texttt{circFact} \stackrel{\text{def}}{=} x:=\lambda z.\texttt{if}\ z=0\ \texttt{then}\ 1\ \texttt{else}\ z\times(!x)(z-1)$$

It is easy to see that, after executing `circFact`, $(!x)n$ returns the factorial of n. In more detail, the state after executing `circFact` may be informally described thus: *x stores a procedure which computes the factorial of its argument using a procedure stored in x: that procedure should calculate the factorial, and x does store that procedure*. Note an inherent circularity of this description — How can we logically describe such a behaviour, and how can we derive it compositionally?

As the third example, let us consider the following program.

$$\texttt{scheduler} \stackrel{\text{def}}{=} \texttt{map}\ (\lambda y^{(\alpha\Rightarrow\mathsf{Unit})\times\alpha}.(\pi_1(y)(\pi_2(y))))$$

where `map` is the standard higher-order map function:

$$\begin{array}{l}\texttt{map} \stackrel{\text{def}}{=} \lambda f^{X\Rightarrow Y}.\mu m^{\mathsf{List}(X)\Rightarrow\mathsf{List}(Y)}.\lambda l^{\mathsf{List}(X)}. \\ \qquad\qquad \texttt{case}(l)\ \texttt{of}\ \mathsf{Nil}\Rightarrow\mathsf{Nil}\ []\ x::y\Rightarrow(fx)::(my)\end{array}$$

Above $x::y$ is the list whose head is x and whose tail is y. The program `scheduler` receives a list of jobs, where each job is a pair of a function and its argument, and executes these functions with corresponding arguments sequentially. Assuming each function may have side effects, what would be the specification of the scheduler, parameterised by properties of stored programs, and how can we derive it from the program text? A possible informal description would be: *Given a list of jobs* $\langle f_1,x_1\rangle,\langle f_2,x_2\rangle,..,\langle f_{n-1},x_{n-1}\rangle$*, if applying* f_1 *to* x_1 *changes the state* σ_1 *to the state* σ_2*, applying* f_2 *to* x_2 *changes the state* σ_2 *to the state* σ_3*, and so on, and finally applying* f_{n-1} *to* x_{n-1} *changes* σ_{n-1} *to* σ_n*, then feeding the scheduler with this job list starting from* σ_1 *will eventually reach the state* σ_n. Compositional reasoning about such a program should treat higher-order functions, recursion, closures and products to derive an intended assertion from a program text. The proposed logic offers a simple language to specify such complex behaviour with precision, combined with syntax directed proof rules for deriving judgements compositionally.

Summary of Technical Results In the following we summarise the main technical results of the paper.

1. Introduction of a compositional program logic for higher-order functions with global state, extending the logic for pure higher-order functions studied in [21, 22], allowing natural descriptions of complex imperative higher-order behaviours and their compositional verification.
2. Study of the semantic foundations of the logic with respect to a naturally defined model. After establishing soundness of the proof rules, sound and complete characterisation of observational equivalence by validity is proved, using a proof method inspired by game semantics [6, 24, 26]. Basic observations on relative completeness of the proof system are also presented at the end.
3. Exploration of the proposed assertional method and its proof rules through reasoning examples, including a sound embedding and extension of Hoare's proof rules for total correctness, as well as an illustration of verifications for three programming examples presented above.

Outline Section 2 illustrates central ideas of the logic informally. Section 3 introduces the target language and the logic. Section 4 illustrates compositional proof rules for the logic. Section 5 establishes soundness of proof rules and logical characterisation of the contextual congruence. Section 6 presents reasoning examples. Section 7 discusses related work, extensions and further topics. The full version [2] presents detailed definitions, more examples and all missing proofs.

2. Preview

This section illustrates the key ideas of assertions for imperative higher-order functions, starting from a brief review of the logic for pure functions presented in [21, 25].

Pure Higher-Order Functions. In the present approach to program logics, behaviour is asserted by naming them. Consider a simple program which computes a doubling function, $N \stackrel{\text{def}}{=} \lambda x^{\mathsf{Nat}}.x+x$, where Nat is the type of natural numbers. If we apply 5 to N, 10 is returned. More generally, the result of applying any natural number to N is always even. To represent these behavioural properties using logical formulae, we do not mention N itself, but rather describe its properties by *naming* it as, say, f. Thus we can write $f \bullet 5 = 10$ as a property of N, named as f. Similarly we can write

$$\forall x^{\mathsf{Nat}}.\mathit{Even}(f \bullet x) \tag{2.1}$$

where $\mathit{Even}(n)$ is the predicate saying n is even (e.g. $\mathit{Even}(x) \stackrel{\text{def}}{=} \exists n.(x = 2 \times n)$). The operator $\bullet$ is left associative and non-commutative, and may be understood as an analogue of application in applicative structures. Formulae may be combined using all the standard logical connectives and quantifiers, just as in Hoare logic.

Using these formulae (ranged over $C, C', D, \ldots$), the judgement of the logic has the following shape.

$$\{C\}M :_f \{C'\}$$

which can be read as: if M, *named as* f, can *rely* on C as the behaviour of an environment, then the program combined with the environment can *guarantee* C'. The name f is called *anchor*. It can be any fresh name not occurring in M and C. An anchor is used to represent M's point of operation, hence of specification. As an example, the specification for N is:

$$\{\mathsf{T}\}N :_f \{\forall x^{\mathsf{Nat}}.\mathit{Even}(f \bullet x)\} \tag{2.2}$$

which says that the program N named f, under the trivial assumption T, satisfies $\forall x^{\mathsf{Nat}}.\mathit{Even}(f \bullet x)$. By having names in assertions, we can compositionally derive a specification which involves a non-trivial assumption on higher-order variables based on a simple operation: when the function f is applied to an argument, the result $f \bullet x$ is *peeled-off* and replaced by a new anchor name u. For example, we can derive

$$\{\forall x^{\mathsf{Nat}}.\mathit{Even}(f \bullet x)\}\, f3 :_u \{\mathit{Even}(u)\}$$

from two smaller specifications (1) $\{\forall x^{\mathsf{Nat}}.\mathit{Even}(f \bullet x)\}\, f :_m \{\forall x^{\mathsf{Nat}}.\mathit{Even}(m \bullet x)\}$ (an instance of the axiom for variable: f named as m satisfies the same predicate as the one assumed for f); and (2) $\{\mathsf{T}\}3 :_m \{m = 3\}$ ("3" named as m satisfies a predicate $m = 3$) and by combining them together as $(\forall x^{\mathsf{Nat}}.\mathit{Even}(f \bullet x) \wedge m = 3) \supset \mathit{Even}(f \bullet m)$ (which is a simple instance of $(A(x,x) \wedge x = y) \supset A(x,y)$, the standard axiom in predicate logic [33, §2.8]). The same framework works for higher-order programs where multiple variables share assumptions.

$$\{\forall x^{\mathsf{Nat}}.\mathit{Even}(f \bullet x)\}\, f3 + f(f5) + 1 :_u \{\mathit{Odd}(u)\} \tag{2.3}$$

where $\mathit{Odd}(n)$ says n is odd. Now by combining two specifications for N in (2.2) and $L \stackrel{\text{def}}{=} f3 + f(f5) + 1$ in (2.3), we arrive at:

$$\{\mathsf{T}\}\texttt{let } f = N \texttt{ in } L :_u \{\mathit{Odd}(u)\} \tag{2.4}$$

where $\texttt{let } f = N \texttt{ in } L$ is encoded as $(\lambda f.L)N$. The property which is guaranteed by N is simply plugged into the assumption for L. This derivation is similar to a composition rule of Hoare logic, where we infer $\{C\}P_1;P_2\{C'\}$ from $\{C\}P_1\{C_1\}$ and $\{C_1\}P_2\{C\}$.

Mutable Higher-Order Functions. The idea of naming behaviours is naturally extended to stateful computation. A typical example is the following specification of a program that reads the number 7 from a global storage cell x and then returns 9.

$$\{!x = 7\}\ 2 + !x :_u \{u = 9 \wedge !x = 7\}$$

where the logical term $!x$ represents dereferencing x [36]. The resulting behaviour is located at the anchor name u. The assertion says: the program $2 + !x$ returns 9 whenever x initially stores 7, and it does not change this content of x. As another example, this time with side-effects:

$$\{!x = 3\}\ x := (2 + !x)\ ;\ !x :_u \{u = 5 \wedge !x = 5\}$$

where ";" is sequential composition (encodable into call-by-value application).

We now move to assertions describing more complex behaviour where functions cause side-effects during evaluation. Let $W \stackrel{\text{def}}{=} \lambda x.(w := (1 + !w)\,; x + x)$, slightly modified from $\lambda x.x + x$ in the previous paragraph. Recalling L from (2.3), we further let $M \stackrel{\text{def}}{=} \texttt{let } f = W \texttt{ in } L$. Now this function not only satisfies $\mathit{Odd}(u)$, but also *changes a memory cell* w when invoked. Hence we would expect the following:

$$\{!w = 0\}\ \texttt{let } f = W \texttt{ in } L :_u \{\mathit{Odd}(u) \wedge !w = 3\} \tag{2.5}$$

How can we specify the behaviour of W to reach (2.5)? A simple method is to attach pre and post-conditions to invocations of functions by an argument, and assert them as a single predicate. Thus we write:

$$\{C\}\ f \bullet x \searrow u\ \{C'\}$$

This assertion reads: if the state of memory and the environment satisfy C, the invocation of f with an argument

x yields a value named u and a final state, together satisfying C'. "$\searrow$" indicates the evaluation of $f \bullet x$ resulting in u, which is asymmetric unlike the equality $e = e'$. This is due to the non-reversibility of state change. Based on this idea, W named as f has the following specification:

$$EvenS(w,f) \stackrel{\text{def}}{=} \forall x.\forall i.\{!w = i\}\ f \bullet x \searrow u\ \{Even(u) \wedge !w = i+1\}$$

$EvenS(w,f)$ specifies a procedure which, when invoked, would not only increment w but also return an even number: if f is called when $!w = 0$, then f's return value is even and $!w = 1$. In fact from $EvenS(w,f)$ we can derive:

$$\forall x.\{!w = 0\}\ f \bullet x \searrow u\ \{Even(u) \wedge !w = 1\}$$

using standard axioms for universal quantification. Now assume f satisfies the above specification. Then we can derive the following judgement.

$$\{EvenS(w,f) \wedge !w = 0\}\ f\ 5 :_v \{Even(v) \wedge !w = 1\}$$

The key idea here is that when the function (named f) is applied to the argument 3, not only is the result u replaced by a new anchor v, but we also *split* the assumption ($EvenS(w,f)$) in two pieces, its pre-condition C added to the pre-condition of the judgement and its post-condition C' in the post-condition of the judgement. Repeating this, we can derive (2.5) in a compositional way, using essentially the same `let`-rule as for stateless functions.

When working with higher-order functions, assertions with pre/post-conditions can be nested repeatedly and may appear in the pre/post-conditions of other assertions. For example, let $V \stackrel{\text{def}}{=} \lambda y.(!x)y$. If x stores a function with side effects, like W above, then calling V may involve writing to memory. Thus we may assert:

$$\{\mathsf{T}\}\lambda y.(!x)y :_u \{\ \{EvenS(w,!x) \wedge !w = 0\}\ u \bullet n \searrow z\,\{Even(z) \wedge EvenS(w,!x) \wedge !w = 1\}\ \}$$

which says: if V is applied to a natural number n under the condition that x stores a function which satisfies $EvenS(w,u)$, and w initially stores 0, then resulting term evaluates to an even number, with w's final state being 1.

A merit of the present approach in comparison with existing methods is that it can directly assert on the combined behaviour of two or more (possibly higher-order) procedures. For example, consider the following program:

$$M \quad \stackrel{\text{def}}{=} \quad \lambda x^{\mathsf{Nat}}.\ (y := x\ ;\ g(f)\ ;\ g(f)\ ;\ !y + 1) \quad (2.6)$$

where f and g are of types $\mathsf{Unit} \Rightarrow \mathsf{Unit}$ and $(\mathsf{Unit} \Rightarrow \mathsf{Unit}) \Rightarrow \mathsf{Unit}$, respectively. Assume we *only* know the abstract property of f and g which says: if we apply f to g, then the content of y changes its parity, i.e. if it is initially even then it becomes odd and vice versa. The proposed logic formally describes this property as follows, omitting return values:

$$\{Odd(!y)\}\ f \bullet g\,\{Even(!y)\}\ \wedge\ \{Even(!y)\}\ f \bullet g\,\{Odd(!y)\}.$$

Let us denote the above assertion by $A(fg)$. Then a property of M may be asserted as:

$$\{A(fg)\}M :_u \{\forall x^{\mathsf{Nat}}.\{Even(x)\}u \bullet x \searrow z\{Odd(z) \wedge Even(!y)\}\} \quad (2.7)$$

which says: under the assumption about f and g as given, if the argument is even, then the result is odd and the content of y is even. Let the above post-condition be $Even\_then\_Odd(u,y)$. From the same pre-condition, we can also infer the dual property $Odd\_then\_Even(u,y) \stackrel{\text{def}}{=} \forall x^{\mathsf{Nat}}.\{Odd(x)\}u \bullet x \searrow z\{Even(z) \wedge Odd(!y)\}$ or even a conjunction of the two, $Even\_then\_Odd(u,y) \wedge Odd\_then\_Even(u,y)$, as its post-condition. This specification relies on the property of the combined behaviour of f and g and demonstrates a practical benefit of having named higher-order procedures and specifications of their behaviour as an integral part of assertions: we can transparently specify and reason about the complex interplay among two or more procedures which may call each other and which as a whole demonstrate a specific behaviour of interest. Further examples will be treated in § 6, after formally introducing the logic and proof rules.

3. Logic for Imperative Call-by-Value PCF

3.1. Imperative PCF

This subsection briefly reviews the programming language we use, call-by-value PCF with unit, sums and products, augmented with imperative variables (henceforth often called *references*). The grammar of programs is standard [41], given below. We assume an infinite set of *variables*, also called *names*, ranged over by $x, y, z, \ldots$.

(value)

$$\begin{array}{rcl} V,W & ::= & \mathtt{c} \mid x \mid \lambda x^{\alpha}.M \mid \mu f^{\alpha \Rightarrow \beta}.\lambda y^{\alpha}.M \\ & \mid & \langle V,W\rangle \mid \mathtt{in}_i(V) \end{array}$$

(program)

$$\begin{array}{rcl} M,N & ::= & V \mid MN \mid x := N \mid !x \\ & \mid & \mathtt{op}(\vec{M}) \mid \pi_i(M) \mid \langle M,N\rangle \mid \mathtt{in}_i(M) \\ & \mid & \mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 \\ & \mid & \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i\in\{1,2\}} \end{array}$$

The grammar uses types $(\alpha, \beta, \ldots)$, which are given later. Constants are ranged over by $\mathtt{c}$. Examples include the unit $()$, natural numbers $\mathtt{n}$ and booleans $\mathtt{b}$ (either $\mathtt{f}$ or $\mathtt{t}$). $\mathtt{op}(\vec{M})$ (where $\vec{M}$ is a vector of programs) is a standard n-ary arithmetic or boolean operation, such as $+$, $-$, $\times$, $=$ (equality of two numbers/booleans), $\neg$ (negation), $\wedge$ and $\vee$. $!x$ dereferences x while $x := N$ is assignment. All these constructs are standard, cf. [16, 41].

The dynamics of programs is given by the call-by-value reduction relation, using store [16, 41]. A *store* $(\sigma, \sigma', \ldots)$ is a finite map from imperative variables to values. We write $\sigma[x \mapsto V]$ for the store which maps x to V

and otherwise agrees with δ. The call-by-value reduction, written $(M,\sigma) \longrightarrow (M',\sigma')$, is standard [16, 41]. We only list the rules for assignment and dereference. Below $\sigma(x)$ and $\sigma[x \mapsto V]$ indicate $x \in \mathsf{dom}(\sigma)$.

$$(!x,\ \sigma) \ \to\ (\sigma(x),\ \sigma) \qquad (x := V,\ \sigma) \ \to\ ((),\ \sigma[x \mapsto V])$$

We also write $(M,\sigma) \Downarrow (V,\sigma')$ for $(M,\sigma) \to^* (V,\sigma')$, and $M \Downarrow V$ for $(M,\emptyset) \to^* (V,\emptyset)$.

The grammar of types is also standard [16, 41].

$$\begin{array}{rcl} \alpha,\beta & ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \alpha \Rightarrow \beta \mid \alpha \times \beta \mid \alpha + \beta \\ \rho & ::= & \alpha \mid \mathsf{Ref}(\alpha) \end{array}$$

We call $\alpha,\beta,\ldots$ *value types*, and $\mathsf{Ref}(\alpha),\ldots$ *reference types*. Reference types are restricted to carrying only non-reference types (called Reduced ML in [42]). Lifting this restriction leads to a distinct class of behaviour which deserves a logical treatment in its own right, see § 7 for further discussions on these extensions. Note a reference can still carry arbitrary higher-order procedures. A *basis* is a finite map from names to types. $\Gamma,\Gamma'\ldots$ range over bases whose codomains are value types, while $\Delta,\Delta',\ldots$ range over bases whose codomains are reference types. $\mathsf{dom}(\Gamma)$ (resp. $\mathsf{dom}(\Delta)$) denotes the domain of Γ (resp. of Δ). The typing rules are standard [41] and omitted. We write $\Gamma;\Delta \vdash M : \alpha$ when M has type α under Γ and Δ, with $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$.

3.2. Terms, Formulae and Judgement

Terms and Formulae. The logical language is that of first-order logic with equality [33, § 2.8] augmented with an assertion for the evaluation of stateful expressions. The grammar of terms and formulae follows.

$$\begin{array}{rcl} e & ::= & x^{\alpha} \mid () \mid \mathbf{c} \mid \mathbf{op}(\vec{e}) \\ & \mid & \langle e,e' \rangle \mid \pi_i(e) \mid \mathsf{inj}_i^{\alpha+\beta}(e) \mid !(x^{ref(\alpha)}) \\ C & ::= & e = e' \mid \neg C \mid C \wedge C' \mid C \vee C' \mid C \supset C' \\ & \mid & \forall x^{\alpha}.C \mid \exists x^{\alpha}.C \mid \{C\}\ e \bullet e' \searrow x\ \{C'\} \end{array}$$

The first set of expressions $(e,e',\ldots)$ are *terms* while the second set are *formulae* $(A,B,C,\ldots)$. Terms, which are from [21, 25] except $!x$, include all the constants $(\mathbf{c},\mathbf{c}',..)$ and first-order operations of the target programming language. We also have a paring, projection and injection operation. $!x$ denotes the dereference of x.

The predicate $\{C\}\ e \bullet e' \searrow x\ \{C'\}$ is called *evaluation formula*, where the name x binds its free occurrences in C'. Intuitively, $\{C\}\ e \bullet e' \searrow x\ \{C'\}$ asserts that an invocation of e with an argument e' under the (hypothetical) initial state C terminates with a final state and a resulting value, named as u, both described by C'. $\bullet$ is non-commutative. $\mathsf{fv}(e)$ denotes the free variables occurring in e. We define two kinds of capture-avoiding substitutions $C[e/x]$ and $C[e/!x]$, see [2, § 3.3].

Terms and formulae are typed starting from type-annotated variables. Two names in a formula should have the same type, similarly for a pair of equated terms. $!(x^{\rho})$ is typed as α iff $\rho = \mathsf{Ref}(\alpha)$. If e_1, e_2 and z are typed as $\alpha \Rightarrow \beta$, α and β, respectively, then $\{C\}\ e_1 \bullet e_2 \searrow z\ \{C'\}$ is well-typed. The remaining well-typedness conditions are naturally given [2]. A boolean typed term is also used as a formula. Hereafter we only consider well-typed terms and formulae and often omit type annotations. We shall write $\Theta \vdash C$ if C is well-typed with its free names typed following Θ, where $\Theta,\Theta',\ldots$ combine two kinds of bases.

Convention 1 $C_1 \equiv C_2$ stands for $(C_1 \supset C_2) \wedge (C_2 \supset C_1)$ (the logical equivalence of C_1 and C_2). We use truth T (definable as $1 = 1$) and falsity F (which is $\neg\mathsf{T}$). The standard binding convention is always assumed. $\mathsf{fv}(C)$ denotes the set of *free variables* in C. $\{C\}\ e_1 \bullet e_2 \searrow e'\ \{C'\}$ with e' not a variable, stands for $\{C\}\ e_1 \bullet e_2 \searrow x\ \{x = e' \wedge C'\}$ with x fresh; and $\{C\}\ e_1 \bullet e_2\ \{C'\}$ for $\{C\}\ e_1 \bullet e_2 \searrow ()\ \{C'\}$. Formulae are often called *assertions*.

Some small examples: $y = 6$ is an assertion which says y is equal to 6; $!y = 6$ says the content of a memory cell y is equal to 6. $C \stackrel{\text{def}}{=} \forall i,n.\ \{!w = n\}\ !x \bullet i \searrow 2 \times i\ \{!w = n+1\}$ says x stores a function which, when invoked, increments w and returns the double of the argument. This is satisfied when, for example, $f(w) \stackrel{\text{def}}{=} \lambda z.(w := !w+1; z \times 2)$ is stored in x. $D \stackrel{\text{def}}{=} \{C \wedge !w{=}0\}u \bullet 3 \searrow 6\{C \wedge !w{=}1\}$ says that, if u is invoked with 3 in a state satisfying $!w{=}0$ as well as C, then the returned value is 6 and the final state is $!w{=}1$. This is satisfied by $\lambda y.(!x)y$ named u, with x storing $f(w)$ above.

Judgement. Following Hoare [19], a judgement in the present program logic consists of a pair of formulae and a program, which takes the following shape.

$$\{C\}\ M^{\Gamma;\Delta;\alpha}\ :_u\ \{C'\}$$

This sequent is used for both validity and provability. If we wish to be specific, we prefix it with either $\vdash$ (for provability) or $\models$ (for validity). In $\{C\}\ M^{\Gamma;\Delta;\alpha}\ :_u\ \{C'\}$, we assume $\Gamma;\Delta \vdash M : \alpha$. u is called the *anchor* of the judgement and should not be in $\mathsf{dom}(\Gamma,\Delta) \cup \mathsf{fv}(C)$. The formula C is the *pre-condition*; and C' is the *post-condition*. We say $\models \{C\}\ M^{\Gamma;\Delta;\alpha}\ :_u\ \{C'\}$ is *well-typed* if (1) $\Gamma;\Delta \vdash M : \alpha$; and (2) $\Gamma,\Delta,\Theta \vdash C$ and $u : \alpha, \Gamma,\Delta,\Theta \vdash C'$ for some Θ such that $\mathsf{dom}(\Theta) \cap (\mathsf{dom}(\Gamma,\Delta) \cup \{u\}) = \emptyset$. The same condition applies to judgements on provability. Then the *primary names* in this well-formed judgement are $\mathsf{dom}(\Gamma,\Delta) \cup \{u\}$. The *auxiliary names* in this judgement are those free names in C and C' which are not primary (for example, in "$\{x = i\}2 \times x :_u \{u = 2 \times i\}$", x and u are primary while i is auxiliary; u is in addition its

anchor). We often omit the typing of a program from a judgement, writing $\{C\}\ M :_u \{C'\}$.

Intuitively, $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ says that: *if M is closed by values satisfying C (for functional variables), and is evaluated starting from a store satisfying C (for imperative variables), then it terminates with a value named u and final state, satisfying C'.* This informal reading will be made precise in § 5.1.

3.3. Axioms and Rules for Validity

When using the proof rules presented in the next section, one often needs to calculate validity of assertions. Formally, a formula C is *valid* if it is true under arbitrary models (the class of models we consider will be discussed in § 5). Practically, we can often calculate validity syntactically. As is standard [19], we shall freely use axioms, rules and theorems from propositional calculus, first-order logic with equality [33, §2.8], and formal number theory. There are also natural axioms for data types, such as $C[()/x^{\mathsf{Unit}}] \equiv C$, $\mathtt{in}_i(e) = \mathtt{in}_j(e') \supset i = j \wedge e = e'$ etc. Further, the following axioms for evaluation formulae are often useful (see [2] for more axioms). Below and henceforth $A, B, \ldots$ denote *stateless formulae*, where a formula is *stateless* if a name of a reference type occurs only inside the pre/post conditions of evaluation formulae. $C^{\tilde{}\vec{x}}$ is C in which no name from $\vec{x}$ freely occurs.

(e1) $\{C_1\}\, e_1 \bullet e_2 \searrow z\{C\} \ \wedge\ \{C_2\}\, e_1 \bullet e_2 \searrow z\{C\}$
$\equiv\ \{C_1 \vee C_2\}\, e_1 \bullet e_2 \searrow z\,\{C\}$

(e2) $\{C\}\, e_1 \bullet e_2 \searrow z\{C_1\} \ \wedge\ \{C\} e_1 \bullet e_2 \searrow z\{C_2\}$
$\equiv\ \{C\}\, e_1 \bullet e_2 \searrow z\,\{C_1 \wedge C_2\}$

(e3) $\{\exists x^{\alpha}.C\}\, e_1 \bullet e_2 \searrow z\,\{C'^{\tilde{}x}\} \ \equiv\ \forall x^{\alpha}.\{C\}\, e_1 \bullet e_2 \searrow z\,\{C'\}$

(e4) $\{C^{\tilde{}x}\}\, e_1 \bullet e_2 \searrow z\,\{\forall x^{\alpha}.C'\} \ \equiv\ \forall x^{\alpha}.\{C\}\, e_1 \bullet e_2 \searrow z\,\{C'\}$

(e5) $\{A \wedge C\}\, e_1 \bullet e_2 \searrow z\,\{C'\} \ \equiv\ A \supset \{C\} e_1 \bullet e_2 \searrow z\{C'\}$

(e6) $\{C\} e_1 \bullet e_2 \searrow z\{C'\} \ \supset\ \{C \wedge A\} e_1 \bullet e_2 \searrow z\{C' \wedge A\}$

(e7) $\{C_0\} e_1 \bullet e_2 \searrow z\{C_0'\} \supset \{C\}\, e_1 \bullet e_2 \searrow z\,\{C'\}$
when $C \supset C_0$ and $C_0' \supset C$

4. Proof Rules

The proof rules are given in Figure 1. In each rule, we use the notational conventions from the preceding sections. We assume all occurring judgements are well-typed, and no primary names in the premise(s) occur as auxiliary names in the conclusion. Below we illustrate key aspects of these rules.

[*Var, Const*] say that, if we wish to assert C about a datum named u, we should assume the same property, with the datum substituted for u.

[*Add*] is the rule for the addition operator, which assumes the left-to-right evaluation order, indicating both the state change induced by evaluation and the resulting values. Similarly for other first-order operators.

[*Abs*] says: if we know, under the assumptions A (which does not talk about x) and starting from C, evaluation of M always terminates with a result m and a state which together satisfy C', then we can guarantee $\lambda x.M$ named u satisfies the same property under the same assumption A, now presented as an evaluation formula replacing M with a call to u by x, $u \bullet x$, with the result m.

[*App*] says: if we know M reaches C_0 starting from C, and N reaches C_1 starting from C_0, and, moreover, we know putting them together and applying them reaches C' starting from C_1, then MN reaches C' starting from u.

[*If, In*$_1$*, Case, Pair, Proj*$_1$]** are natural rules for standard data types, similar to **[*Add*]**.

[*Deref*] is understood as **[*Var, Const*]**. If we wish to have C for a program $!x$ named u, then we should assume the same thing for the content of x, substituting $!x$ for u.

[*Assign*] uses two substitutions $C'[m/!x][()/u]$. The notation $[m/!x]$ stands for replacing all occurrences of $!x$

Figure 1 Proof Rules

$$[Var]\ \frac{-}{\{C[x/u]\}\ x :_u \{C\}} \qquad [Const]\ \frac{-}{\{C[\mathbf{c}/u]\}\ \mathtt{c} :_u \{C\}}$$

$$[Add]\ \frac{\{C\}\, M_1 :_{m_1} \{C_0\} \quad \{C_0\}\, M_2 :_{m_2} \{C'[m_1 + m_2/u]\}}{\{C\}\, M_1 + M_2 :_u \{C'\}}$$

$$[Abs]\ \frac{\{C \wedge A^{\tilde{}x}\}\, M :_m \{C'\}}{\{A\}\ \lambda x.M :_u \{\forall x.\{C\}\, u \bullet x \searrow m\,\{C'\}\}}$$

$$[App]\ \frac{\{C\}\, M :_m \{C_0\} \quad \{C_0\}\, N :_n \{\, C_1 \wedge \{C_1\}\, m \bullet n \searrow u\,\{C'\}\,\}}{\{C\}\, MN :_u \{C'\}}$$

$$[If]\ \frac{\{C\}\, M :_b \{C_0\} \quad \{C_0[\mathbf{t}/b]\}\, M_1 :_u \{C'\} \quad \{C_0[\mathbf{f}/b]\}\, M_2 :_u \{C'\}}{\{C\}\ \mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u \{C'\}}$$

$$[In_1]\ \frac{\{C\}\, M :_v \{C'[\mathtt{in}_1(v)/u]\}}{\{C\}\ \mathtt{in}_1(M) :_u \{C'\}}$$

$$[Case]\ \frac{\{C^{\tilde{}\vec{x}}\}\, M :_m \{C_0^{\tilde{}\vec{x}}\} \quad \{C_0[\mathtt{in}_i(x_i)/m]\}\, M_i :_u \{C'^{\tilde{}\vec{x}}\}}{\{C\}\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}}$$

$$[Pair]\ \frac{\{C\}\, M_1 :_{m_1} \{C_0\} \quad \{C_0\}\, M_2 :_{m_2} \{C'[\langle m_1, m_2\rangle/u]\}}{\{C\}\ \langle M_1, M_2\rangle :_u \{C'\}}$$

$$[Proj_1]\ \frac{\{C\}\, M :_m \{C'[\pi_1(m)/u]\}}{\{C\}\ \pi_1(M) :_u \{C'\}}$$

$$[Deref]\ \frac{-}{\{C[!x/u]\}\ !x :_u \{C\}} \qquad [Assign]\ \frac{\{C\}\, M :_m \{C'[m/!x][()/u]\}}{\{C\}\ x := M :_u \{C'\}}$$

$$[Rec]\ \frac{\{A^{\tilde{}x} \wedge \forall j \leq i.B(j)[x/u]\}\ \lambda y.M :_u \{B(i)\}}{\{A\}\ \mu x.\lambda y.M :_u \{\forall i.B(i)\}}$$

$$[Promote]\ \frac{\{A\}\ V :_u \{B\}}{\{A \wedge C\}\ V :_u \{B \wedge C\}}$$

$$[Consequence]\ \frac{C \supset C_0 \quad \{C_0\}\, M :_u \{C_0'\} \quad C_0' \supset C'}{\{C\}\, M :_u \{C'\}}$$

by m, while $[()/u]$ is the standard substitution of $()$ for u. The first substitution $C'[m/!x]$ says the result of the assignment $x := M$ is turning what is stated about m in $C'[m/!x]$ into the property of $!x$. The second substitution $[()/u]$ says, in effect, the assignment command terminates (because $()$ is the unique value of type Unit).

[*Rec*] is for the total correctness of recursion [21, 25]. It is based on mathematical induction, though by choosing an appropriate domain and a well-ordering, we can extend the rule to well-founded induction. We later show how this rule can cleanly and precisely encode (and extend) known proof rules for total correctness of the while loop and recursive procedures.

[*Promote*] extends the stateless pre/post-conditions to general ones by conjunction. The rule is sound because a value does not (immediately) cause state change.

[*Consequence*] The rule follows the standard consequence rule in Hoare logic. Checking the validity of entailments is in general intractable, though in practice one can often appeal to syntactic reasoning, cf. §3.3. Other structural rules are discussed in [2, §4].

5. Soundness and Observational Completeness

5.1 Models and Soundness

We first introduce models for assertions and judgements. Their operational nature has the merit of faithfulness to programs' behaviours (e.g. a predicate claiming the existence of unrealisable functions is unsatisfiable); extensibility (e.g. polymorphisms); and conciseness. Different models are possible: [23] constructs a uniform universe of models from typed processes.

The semantics centres on programs which do not own free non-reference variables.

Definition 1 (semi-closed programs [35]) $\Gamma;\Delta \vdash M : \alpha$ is *semi-closed* (resp. *closed*) when $\mathsf{dom}(\Gamma) = \emptyset$ (resp. $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Delta) = \emptyset$), often written $\Delta \vdash M : \alpha$ (resp. $\vdash M : \alpha$).

Let $\cong$ be the standard observational congruence for the imperative PCFv [16], based on convergence to semi-closed values. An *abstract value of type* $\Delta;\alpha$ is a $\cong$-congruence class of semi-closed values typed α under Δ. We write $[V]^{\Delta;\alpha}$ for an abstract value represented by $\Delta \vdash V : \alpha$. A model is defined using abstract values.

Definition 2 (models) A *model of type* $\Gamma;\Delta$ is a pair (ξ,σ) such that ξ is a finite map from $\mathsf{dom}(\Gamma)$ to abstract values such that each $x \in \mathsf{dom}(\Gamma)$ is mapped to an abstract value typed as $[V]^{\Delta;\Gamma(x)}$; and σ is a finite map from $\mathsf{dom}(\Delta)$ to abstract values such that each $x \in \mathsf{dom}(\Delta)$ is mapped to an abstract value typed as $[V]^{\Delta;\mathsf{Ref}(\alpha)}$ with $\Delta(x) = \mathsf{Ref}(\alpha)$. We let $\mathcal{M}, \ldots$ range over models.

We write $\Gamma;\Delta \vdash \mathcal{M}$ when $\mathcal{M}$ is a model of type $\Gamma;\Delta$. Intuitively, ξ and σ in (ξ,σ) respectively denote a standard functional environment and a store, taken modulo $\cong$.

Assume given a formula C and a model $\mathcal{M}$, both typed under $\Gamma;\Delta$. Then each term in C is inductively interpreted under $\mathcal{M}$ as an abstract value in the obvious way, except each name of a reference type is interpreted as that name itself (so that, in effect, we treat reference-typed names as constants). As examples, given $\mathcal{M} = (\xi,\sigma)$, a functional variable x^α is interpreted as $\xi(x)$; dereferencing $!y$ is interpreted as $\sigma(y)$; and a pair $\langle e, e'\rangle$ is interpreted as a pair of abstract values interpreting e and e' (the full definition is found in [2, 25, §4.3]). We write $[\![e]\!]\mathcal{M}$ for the interpretation of e under $\mathcal{M}$.

The satisfaction relation is defined using two disjoint models, one interpreting primary names and another auxiliary names. Writing $I, I', \ldots$ for models used for interpreting auxiliary names, the satisfaction relation is written:

$$\mathcal{M}^{\Gamma;\Delta} \models^I C$$

which reads: *under I, C is satisfied by $\mathcal{M}$*. The satisfaction relation is defined following the standard first-order logic with equality [33, §2.8] with the equality predicate interpreted as the identity relation, adding the following clause for evaluation formulae. Let $\mathcal{M} = (\xi,\sigma_0)$. $\Downarrow$ is defined from that of concrete programs. We define: $\mathcal{M}^{\Gamma;\Delta} \models^I \{C\} e_1 \bullet e_2 \searrow x\{C'\}$ if

$$\begin{array}{l}\forall\sigma.\ (\ \Delta \vdash \sigma \wedge (\xi,\sigma) \models^I C \\ \quad \supset\ \exists V,\sigma'.\ ([\![e_1]\!]\mathcal{M}\cdot I[\![e_2]\!]\mathcal{M}\cdot I,\ \sigma) \Downarrow ([V]^{\Delta;\beta},\sigma') \\ \qquad \text{such that } (\xi \cup x{:}[V]^{\Delta;\beta},\sigma') \models^I C'\ )\end{array}$$

The left-hand side says: if the interpretation of e_1 is invoked with that of e_2 as an argument, then for any state σ satisfying C, the invocation starting from σ will converge with a value named x and a state σ', together satisfying C'. As the first result, we formally justify axioms in § 3.3 as the basis of reasoning. For the proof, see [2, §5.2].

Proposition 1 (soundness of axioms) *All axioms in § 3.3 are valid under arbitrary (well-typed) models.*

We are now ready to formalise the semantics of judgements. Below $M\xi$ denotes the substitution of values following ξ, confusing abstract values and concrete values.

Definition 3 (semantics of judgement) $\models \{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ iff, for each I, ξ and σ, whenever $(\xi,\sigma) \models^I C$, we have $(M\xi,\sigma) \Downarrow (V,\sigma')$ such that $(\xi \cdot u{:}[V]^{\Delta;\alpha},\sigma') \models^I C'$.

We conclude this subsection with the main theorem of the paper, proved in [2, §5.3].

Theorem 1 (soundness of proof rules) *If* $\vdash \{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ *then* $\models \{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\}$.

5.2 Observational Completeness

Compositional semantics dictates that programs with the same contextual behaviour are in principle interchangeable without affecting the observable behaviour of whole software, thus offering foundations for modular software engineering. Compositional program logics extend this idea by further allowing programs with the same specifications to be interchangeable without affecting the observable behaviour of the whole, up to a required specification. For this to be materialised, it is essential that valid assertions for programs capture precisely the contextual behaviour of programs [18, 34, 35]. This criterion may be stated with different degrees of exactness:

1. Are two programs contextually equivalent if and only if they satisfy the same set of assertions? That is, are $M_1 \cong M_2$ if and only if, for each A, $u : M_1 \models A$ implies $u : M_2 \models A$ and vice versa?
2. For each program, is there an assertion (*characteristic formula*) which fully describes its behaviour? That is, for each M, can we find A such that $u : M \models A$ and $u : N \models A$ implies $M \cong N$?

Clearly (2) entails (1). Further, these questions can also be asked at the level of provability. (1) may be regarded as an essential property of any program logic which aims to capture observable behaviour of programs. The following establishes (1) for our logic. For (2) (including its provability version), see § 7.

For establishing (1), we proceed as follows.

Step 1: We introduce a variant of *finite canonical forms* (FCFs) [6, 24, 26] which represent a limited class of behaviours and whose properties are, therefore, more readily extracted.

Step 2: We show characteristic formulae of FCFs w.r.t. total correctness are derivable using our proof rules.

Step 3: By reducing a differentiating context of two terms to FCFs and further to their characteristic formulae, we show any semantically distinct programs can be differentiated by an assertion, leading to the characterisation of $\cong$ by validity.

For our present purpose, it suffices to focus on the following class of assertions. Below $\sqsubseteq$ is the standard contextual ordering.

Definition 4 An assertion C is a *total correctness assertion (TCA) at* u if whenever $(\xi \cdot u : \kappa, \sigma) \models^I C$ and $\kappa \sqsubseteq \kappa'$, we have $(\xi \cdot u : \kappa', \sigma) \models^I C$.

Intuitively, total correctness is a property which is closed upwards — if a program M satisfies one and there is a more defined program N then N also satisfies it, see [2, §6]. The notion of characteristic formulae needs be refined for total correctness:

Definition 5 (characteristic formulae) Given a semi-closed V, a TCA C *characterises* V iff: (1) $\models \{\mathsf{T}\}V^{\Delta;\alpha} :_u \{C\}$ and (2) $\models \{\mathsf{T}\}W^{\Delta;\alpha} :_u \{C\}$ implies $V \sqsubseteq W$.

We now introduce FCFs. Henceforth we only consider Nat and arrow types for simplicity. This does not influence the arguments. Finite canonical forms (FCFs), ranged over by $F, F', \ldots$, are a subset of typable terms given by the following grammar (with obvious translations). $U, U', \ldots$ range over FCFs which are values.

$$F ::= \mathtt{n} \mid \omega^\alpha \mid \lambda x.F \mid \mathtt{case}\, x\, \mathtt{of}\, \langle \mathtt{n_i} : F_i \rangle_{n_i \in X} \mid x := U; F \mid \mathtt{let}\, x = yU\, \mathtt{in}\, F \mid \mathtt{let}\, x = !y\, \mathtt{in}\, F$$

where in the case construct, X is a finite non-empty subset of natural numbers (it diverges for others); and ω^α stands for a diverging closed term of type α. We also set $\Omega^{\alpha \Rightarrow \beta} \stackrel{\text{def}}{=} \lambda x^\alpha.\omega^\beta$. We omit the obvious induced typing rules. In the functional sublanguage, FCFs represent essentially finite behaviour. Here we use FCFs for their tractability to derive characteristic formulae.

Proposition 2 *For each semi-closed* $\Delta \vdash U : \alpha$*, we have* $\vdash \{\mathsf{T}\}U^{\Delta;\alpha} :_u \{C\}$ *such that* C *characterises* U.

The proof uses derived proof rules tailored for extracting strongest postconditions from FCFs, for which we inductively prove the property in Definition 5. Then a strongest postcondition of T w.r.t. U gives the desired formula, see [2, §6]. Note Proposition 2 implies (relative) completeness of $\vdash$ for FCFs w.r.t. total correctness.

Write $\Gamma;\Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$ when $\models \{C\}M_1^{\Gamma;\Delta;\alpha} :_u \{C'\}$ iff $\models \{C\}M_2^{\Gamma;\Delta;\alpha} :_u \{C'\}$. The main result follows.

Theorem 2 (observational completeness) $\Gamma;\Delta \vdash M_1 \cong M_2 : \alpha$ *if and only if* $\Gamma;\Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$.

PROOF: The "only if" direction is by Definition 3. For the "if" direction, we prove the contrapositive. For brevity we present the reasoning for purely functional sublanguage (the only change for the full language is a slightly more complex context). Suppose $M_1 \ncong M_2$. It suffices [6, 24, 26] to take closed FCFs $\vec{F}$ such that, e.g. $V\vec{F} \Downarrow \mathtt{n}$ but $W\vec{F} \not\Downarrow \mathtt{n}$ (it may either diverge or converge to a different numeral). For simplicity we consider a single such F, and let $[\![F]\!]_f$ be its characteristic formula at f via Proposition 2. We now infer, using Definition 5:

$$\begin{array}{ll} & u : [V], f : [F] \models u \bullet f \searrow \mathbf{n} \\ \supset & \forall \kappa \sqsupseteq [F].\ (u : [V], f : \kappa \models u \bullet f \searrow \mathbf{n}) \\ \supset & \forall \kappa.\ (f : \kappa \models [\![F]\!]_f \supset u : [V], f : \kappa \models u \bullet f \searrow \mathbf{n}) \\ \supset & u : [V] \models \forall f.([\![F]\!]_f \supset u \bullet f \searrow \mathbf{n})\ . \end{array}$$

But $u : [W] \not\models \forall f.([\![F]\!]_f \supset u \bullet f \searrow \mathbf{n})$, hence done. □

One of the significant consequences of Theorem 2 is that a strongest post condition for total correctness of T w.r.t. each semi-closed value (if any), not restricted to FCFs, is its characteristic formula.

Figure 2 Hoare Logic with Recursive Procedure

$$[Skip]\ \frac{}{\Sigma \vdash \{C\}\mathtt{skip}\{C\}} \qquad [AsH]\ \frac{}{\Sigma \vdash \{C[e/u]\}x := e\{C\}}$$

$$[Seq]\ \frac{\Sigma \vdash \{C\}P\{C_0\}\ \ \Sigma \vdash \{C_0\}Q\{C'\}}{\Sigma \vdash \{C\}P;Q\{C'\}}$$

$$[IfH]\ \frac{\Sigma \vdash \{C \wedge e\}P_1\{C'\}\ \ \Sigma \vdash \{C \wedge \neg e\}P_2\{C'\}}{\Sigma \vdash \{C\}\,\mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2\{C'\}}$$

$$[While]\ \frac{\begin{array}{c} C \wedge e \supset e' \gtrsim 0 \\ \Sigma \vdash \{C \wedge e \wedge e' = n\}P\{C \wedge e' \lesssim n\} \end{array}}{\Sigma \vdash \{C\}\,\mathtt{while}\ e\ \mathtt{do}\ P\{C \wedge \neg e\}}$$

$$[Call]\ \frac{\{C\}p\{C'\} \in \Sigma}{\Sigma \vdash \{C\}\,\mathtt{call}\ p\ \{C'\}}$$

$$[RecProc]\ \frac{\begin{array}{c} \Sigma, \{\exists j \lesssim i.C(j)\}p\{C_0\} \vdash \{C(i)\}P\{C_0\} \\ \Sigma, \{\exists i.C(i)\}p\{C_0\} \vdash \{C\}Q\{C'\} \end{array}}{\Sigma \vdash \{C\}\mathtt{proc}\ p = P\ \mathtt{in}\ Q\{C'\}}$$

6. Reasoning Examples

6.1. Deriving and Extending Hoare Logic

We first embed the standard proof rules of Hoare logic for total correctness with recursive procedures [27] in the present logic, establishing a precise connection between the proposed logic and traditional program logics. The syntax of programs ($P, Q, ..$) is the standard while language augmented with argument-free procedures, see Figure 6.1. $p, q, \ldots$ range over procedure labels and e is given as: $e ::= x \mid \mathtt{c} \mid !x \mid \mathtt{op}(e_1, ..., e_n)$. In $\mathtt{proc}\ p = P\ \mathtt{in}\ Q$, a procedure body P is named p, where we allow calls to p to occur in P.

We consider a logic for total correctness. Formulae, ranged over by $C, C', \ldots$, are a proper subset of the logic for imperative PCF, having only natural numbers as data types and missing evaluation formulae. We also use e as terms in formulae. The judgement takes the shape $\Sigma \vdash \{C\}P\{C'\}$, where $\{C\}P\{C'\}$ is the standard Hoare triple and Σ is a finite map from procedural labels to triples, each taking the form $\{C\}p\{C'\}$. Figure 6.1 presents the standard proof rules [27]. We also use the standard consequence rule. The encoding of programs into PCF-terms is standard (procedure labels are simply taken to be names).

$$[\![\mathtt{skip}]\!] \stackrel{\text{def}}{=} () \quad [\![x := e]\!] \stackrel{\text{def}}{=} x := e \quad [\![P;Q]\!] \stackrel{\text{def}}{=} [\![P]\!];[\![Q]\!]$$
$$[\![\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q]\!] \stackrel{\text{def}}{=} \mathtt{if}\ e\ \mathtt{then}\ [\![P]\!]\ \mathtt{else}\ [\![Q]\!]$$
$$[\![\mathtt{while}\ e\ \mathtt{do}\ P]\!] \stackrel{\text{def}}{=} (\mu w.\lambda().\mathtt{if}\ e\ \mathtt{then}\ [\![P]\!];(w())\ \mathtt{else}\ ())()$$
$$[\![\mathtt{call}\ p]\!] \stackrel{\text{def}}{=} p()$$
$$[\![\mathtt{proc}\ p = P\ \mathtt{in}\ Q]\!] \stackrel{\text{def}}{=} (\lambda p.[\![Q]\!])(\mu p.\lambda().[\![P]\!])$$

All commands have unit type. When M has unit type in $\{C\}M :_u \{C'\}$, we can safely omit the anchor because $C[()/u] \equiv C$. Hence, just like a Hoare triple, hereafter we often write $\{C\}M\{C'\}$. The judgement $\Sigma \vdash \{C\}P\{C'\}$ is translated as $\{[\![\Sigma]\!] \wedge C\}[\![P]\!]\{C'\}$ with $[\![\emptyset]\!] = \mathsf{T}$ and $[\![\Sigma, \{C\}p\{C'\}]\!] \stackrel{\text{def}}{=} [\![\Sigma]\!] \wedge \{C\}p\bullet()\{C'\}$. For the standard assignment rule [*AsH*], we use the following rule derivable from the rules in Figure 1.

$$[Simple]\ \frac{}{\{C[e/u]\}e :_u \{C\}}$$

Then [*AsH*] is decomposed into [*Simple*] and [*Assign*] of Figure 1. For [*RecProc*], the recursion rule in the present logic, [*Rec*], gives a precise account of the induction principle for recursive procedures. Below (e3) etc. are axioms from §3.3, indicating their use in the consequence rule and (Asm) stands for "assumption".

1.	$\{[\![\Sigma]\!] \wedge \{\exists j \lesssim i.C(j)\}p\bullet()\{C'\} \wedge C(i)\}\ [\![P]\!]\ \{C_0\}$	(Asm)
2.	$\{[\![\Sigma]\!] \wedge \forall j^{\mathsf{Nat}}.\{j \lesssim i \wedge C(j)\}p\bullet()\{C'\} \wedge C(i)\}\ [\![P]\!]\ \{C_0\}$	(e3)
3.	$\{[\![\Sigma]\!] \wedge \forall j^{\mathsf{Nat}} \lesssim i.\{C(j)\}p\bullet()\{C'\} \wedge C(i)\}\ [\![P]\!]\ \{C_0\}$	(e5)
4.	$\{[\![\Sigma]\!] \wedge \forall j^{\mathsf{Nat}} \lesssim i.\{C(j)\}p\bullet()\{C'\}\}\ \lambda().[\![P]\!] :_m \{\{C(i)\}m\bullet()\{C_0\}\}$	(Abs)
5.	$\{[\![\Sigma]\!]\}\ \mu p.\lambda().[\![P]\!] :_m \{\forall i.\{C(i)\}m\bullet()\{C_0\}\}$	(Rec)
6.	$\{[\![\Sigma]\!]\}\ \mu p.\lambda().[\![P]\!] :_m \{\{\exists i.C(i)\}m\bullet()\{C_0\}\}$	(e3)
7.	$\{[\![\Sigma]\!] \wedge \{\exists i.C(i)\}p\bullet()\{C_0\} \wedge C\}\ [\![Q]\!]\ \{C'\}$	(Asm)
8.	$\{[\![\Sigma]\!]\}\ \lambda p.[\![Q]\!] :_n \{\forall p.\{\exists i.C(i)\}p\bullet()\{C_0\} \supset \{C\}n\bullet p\{C'\}\}$	(Abs)
9.	$\{[\![\Sigma]\!] \wedge C\}\ (\lambda p.[\![Q]\!])(\mu p.\lambda().[\![P]\!])\ \{C'\}$	(6, 8, App)

In Lines 2 and 3, we use $\exists j \lesssim i.C \stackrel{\text{def}}{=} \exists j.(j \lesssim i \wedge C)$ and $\forall j \lesssim i.C \stackrel{\text{def}}{=} \forall j.(j \lesssim i \supset C)$.

For other rules, [*Skip*] and [*Call*] are immediate; [*Seq*] is from [*App*] and [*Abs*], using $C[()/u] \equiv C$, cf. §3.3; [*IfH*] is by [*Simple*] and [*If*]; [*While*] uses [*Rec*]. See [2] for derivations. Thus, assuming the standard model [33, §3.1] for assertions in Hoare logic:

Theorem 3 (embedding of Hoare logic for total correctness) $\Sigma \vdash \{C\}P\{C'\}$ *implies* $\{[\![\Sigma]\!] \wedge \{C\}\}[\![P]\!]\{C'\}$.

We end this subsection with the extension of the while rule for its use in the imperative PCFv.

$$[While']\ \frac{\begin{array}{c} \{C\}M :_b \{B^b \wedge C\} \quad C \wedge B[\mathbf{t}/b] \supset e' \gtrsim 0 \\ \{C \wedge B[\mathbf{t}/b] \wedge e' = n\}N\{C \wedge e' \lesssim n\} \end{array}}{\{C\}\,\mathtt{while}\ M\ \mathtt{do}\ N\{C \wedge B[\mathbf{f}/b]\}}$$

This and other rules are useful for imperative PCFv, as we shall see in the next subsections.

6.2. Simple Imperative Higher-Order Functions

We further illustrate the use of proof rules with programs which correspond to the assertions in § 2 and § 3.2. Let $Double(u) \stackrel{\text{def}}{=} \forall i.(u \bullet i = i \times 2)$. Then we infer a function with dereference.

1.	$\{Double(!x)\}\ !x :_m \{Double(m)\}$	(Deref)
2.	$\{y = 3\}y :_n \{n = 3\}$	(Var)
3.	$\{Double(!x) \wedge y = 3\}\ (!x)y :_u \{Double(!x) \wedge u = 6\}$	(App)
4.	$\{\mathsf{T}\}\ \lambda y.(!x)y :_u \{\{Double(!x)\}u\bullet 3 \searrow 6\{Double(!x)\}\}$	(Abs)
5.	$\{Double(!x)\}\ (\lambda y.(!x)y)3 :_u \{u = 6 \wedge Double(!x)\}$	(App)

Next we use the following variant of [*Seq*] in § 6.1:

$$[\mathit{Seq'}]\ \frac{\{C\}\,M\,\{C_0\} \quad \{C_0\}\,N :_u \{C'\}}{\{C\}\,M;N :_u \{C'\}}$$

Using [*Seq'*], we can plug-in the post and pre-conditions of the conclusions of $(\lambda y.(!x)y)3$ and $\{\mathsf{T}\}\,x := \lambda z.(z\times 2)\,\{\mathit{Double}(!x)\}$ as:

$$\{\mathsf{T}\}\ x := \lambda z.(z\times 2)\,;\,(\lambda y.(!x)y)3 :_u \{u = 6 \wedge \mathit{Double}(!x)\}$$

By a similar reasoning, we obtain the following which corresponds to C in § 3.2.

$$\{\mathsf{T}\}\ x := \lambda z.(w := !w+1; z\times 2)\ \{\forall i,n.\,\{!w = n\}\,!x \bullet i \searrow 2\times i\,\{!w = n+1\}\}$$

Then similarly, we can derive D in § 3.2.

$$\{\mathsf{T}\}\lambda y.(!x)y :_u \{\{C \wedge\, !w{=}0\}\,u \bullet 3 \searrow 6\,\{C \wedge\, !w{=}1\}\}$$

Combining these by [*Seq'*] gives us:

$$\{C \wedge !w = 0\}\ x := \lambda z.(w := !w+1; z\times 2)\,;\,(\lambda y.(!x)y)3 :_u \{u = 6 \wedge C \wedge\, !w = 1\}$$

Finally we reason for M in (2.6), § 2, page 4, using $A(fg)$ given there.

1. $\{A(fg)\wedge \mathit{Even}(x)\}\ y := x\ \{A(fg)\wedge \mathit{Even}(!y)\}$	(AsH, Conseq)
2. $\{A(fg)\wedge \mathit{Even}(!y)\}\ g(f)\ \{A(fg)\wedge \mathit{Odd}(!y)\}$	(Var×2, App)
3. $\{A(fg)\wedge \mathit{Odd}(!y)\}\ g(f)\ \{A(fg)\wedge \mathit{Even}(!y)\}$	(Var×2, App)
4. $\{A(fg)\wedge \mathit{Even}(!y)\}\ !y+1 :_z \{\mathit{Odd}(z)\wedge \mathit{Even}(!y)\}$	(Simple)
5. $\{A(fg)\}\ M :_u \{\mathit{Even\_then\_Odd}(u,x)\}$	(Seq'×3, Abs)

6.3. Three Programming Examples Revisited

This subsection revisits the examples from the introduction. First, the specification for `closureFact` can be made precise, for example with the following judgement.

$$\{\mathsf{T}\}\ \texttt{closureFact} :_u \{\forall i^{\mathsf{Nat}}.\{\mathsf{T}\}\,u \bullet i\,\{\{\mathsf{T}\}\,!y \bullet () \searrow z\,\{z = i!\}\}\}$$

Next we consider `circFact`, whose specification can be written down as, under $x : \mathsf{Ref}(\mathsf{Nat}\Rightarrow\mathsf{Nat})$:

$$\{\mathsf{T}\}\ \texttt{circFact}\ \{\exists g.(\forall i.\{!x = g\}(!x) \bullet i \searrow i!\{!x = g\} \wedge !x = g)\}$$

The specification says: after executing `circFact`, x stores a procedure which would calculate a factorial if x indeed stores that behaviour itself, and that x does store that behaviour, tersely describing all we need to know about `circFact` including its circularity. For the derivations of these specifications for `closureFact` and `circFact`, see [2].

The following assertion describes `scheduler`'s behaviour ($C(i)$ represents a sequence of states; we assume list operations :: and Nil in the assertion language).

$$\begin{aligned}\mathsf{Sched}(u) \stackrel{\text{def}}{=}\ & \{C(0)\}u \bullet \mathsf{Nil}\{C(0)\} \quad \wedge \\ & \forall g^{\alpha\Rightarrow\mathsf{Unit}}, a^{\alpha}, y^{\mathsf{List}((\alpha\Rightarrow\mathsf{Unit})\times\alpha)} \\ & (\{C(i+1)\}g \bullet a\,\{C(i)\} \wedge \{C(i)\}u \bullet y\,\{C(0)\} \\ & \qquad \supset \{C(i+1)\}u \bullet (\langle g,a\rangle :: y)\,\{C(0)\})\end{aligned}$$

The assertion says: given for example a list $l = [(f,a),(g,b)]$, if we can prove $\{C(0)\}\,fa\,\{C(1)\}$ and $\{C(1)\}\,gb\,\{C(2)\}$, then $\{C(0)\}\,\texttt{scheduler}\,l\,\{C(2)\}$ can be derived. Below we outline how the main judgement

$$\{\mathsf{T}\}\ \texttt{scheduler} :_u \{\mathsf{Sched}(u)\}, \tag{6.1}$$

can be inferred. Setting $\texttt{scheduler} \stackrel{\text{def}}{=} \texttt{map app}$ where $\texttt{app} \stackrel{\text{def}}{=} \lambda z^{(\alpha\Rightarrow\mathsf{Unit})\times\alpha}.(\pi_1(z)(\pi_2(z)))$, we derive:

$$\{\mathsf{T}\}\,\texttt{map} :_m \{\forall f^{\beta\Rightarrow\mathsf{Unit}}\,\mathsf{Map}'(m,f)\} \tag{6.2}$$

$$\{\mathsf{T}\}\,\texttt{app} :_n \{\forall z^{(\alpha\Rightarrow\mathsf{Unit})\times\alpha}.\,\mathsf{App}(n,z)\} \tag{6.3}$$

where:

$$\begin{aligned}\mathsf{Map}'(m,f) \stackrel{\text{def}}{=}\ & \{\mathsf{T}\}m \bullet f \searrow u\{\mathsf{Map}(u,f)\} \\ \mathsf{Map}(u,f) \stackrel{\text{def}}{=}\ & \{C(0)\}u \bullet \mathsf{Nil}\{C(0)\} \quad \wedge \\ & \forall x^{\beta}, y^{\mathsf{List}(\beta)}.(\{C(i+1)\}\,f \bullet x\,\{C(i)\} \ \wedge\ \{C(i)\}\,u \bullet y\,\{C(0)\} \\ & \qquad \supset\ \{C(i+1)\}\,u \bullet (x :: y)\,\{C(0)\}) \\ \mathsf{App}(n,z) \stackrel{\text{def}}{=}\ & (\{C(i+1)\}\,\pi_1(z) \bullet \pi_2(z)\,\{C(i)\} \\ & \qquad \supset\ \{C(i+1)\}\,n \bullet z\,\{C(i)\})\end{aligned}$$

The derivation for `map` in (6.2) follows [25, § 5]. (6.4) is straightforward. We derive (6.1) from (6.2) and (6.4).

1. $\{\mathsf{T}\}\ \texttt{map} :_m \{\forall f.\mathsf{Map}'(m,f)\}$	(6.2)
2. $\{\mathsf{T}\}\ \texttt{app} :_n \{\forall z.\mathsf{App}(n,z)\}$	(6.2)
3. $\{\forall f.\mathsf{Map}'(m,f)\}\ \texttt{app} :_n \{\forall f.\mathsf{Map}'(m,f)\wedge\forall z.\mathsf{App}(n,z)\}$	(Inv)
4. $\{\forall f.\mathsf{Map}'(m,f)\}\ \texttt{app} :_n \{\{\mathsf{T}\}m \bullet n \searrow u\{\mathsf{Sched}(u)\}\}$	(Conseq)
5. $\{\mathsf{T}\}\ (\texttt{map app}) :_u \{\mathsf{Sched}(u)\}$	(1, 4, App)

Line 3 uses the structure rule in [2]. Line 4 uses the following inferences on validity.

$$\begin{aligned}& \forall f.\mathsf{Map}'(m,f) \wedge \forall z.\mathsf{App}(n,z) \\ & \supset \mathsf{Map}'(m,n) \wedge \mathsf{App}(n,z) \quad (\forall\text{-inst}) \\ & \supset \{\mathsf{T}\}m \bullet n \searrow u\{\mathsf{Map}(u,n) \wedge \mathsf{App}(n,z)\} \quad (\text{e6}) \\ & \supset \{\mathsf{T}\}m \bullet n \searrow u\{\mathsf{Map}(u,\langle g,a\rangle) \wedge \mathsf{App}(n,\langle g,a\rangle)\}\ (\forall\text{-inst, (e7)}) \\ & \supset \{\mathsf{T}\}m \bullet n \searrow u\{\mathsf{Sched}(u)\} \quad (\text{modus ponens, (e7)})\end{aligned}$$

7. Further Topics and Related Works

Inferential Completeness As observed in §5.2, our proof system is (relatively) complete for semi-closed FCFs w.r.t. total correctness. Does this extend to the whole language? We believe so in the following sense.

Conjecture. (1) For each semi-closed V, $\{\mathsf{T}\}V :_u \{C\}$ s.t. C characterises V in the sense of Def. 5. (2) For each TCA C', $\models \{\mathsf{T}\}V :_u \{C'\}$ implies $\vdash \{\mathsf{T}\}V :_u \{C'\}$.

The statement says that the assertion language can pin-point, and the proof rules can relatively justify, any

upwards-closed set which has a semi-closed value as its least element.

For partial correctness, the following rule (from [23, §5]) is known to be complete.

$$[\textit{Rec-Partial}]\ \frac{\{A \wedge B[x/u]\}\,\lambda y.M :_u \{B\} \quad B \text{ admissible at } u \quad A \supset \exists x.B}{\{A\}\,\mu x.\lambda y.M :_u \{B\}}$$

where "admissibility" intuitively says that B is about partial correctness [23, §5.3]. The rule can embed and justify known proof rules of loops and recursion for partial correctness, such as the while rule in Hoare logic. Our coming paper will discuss completeness results in detail.

Aliasing and Local State In § 3, it is observed that allowing reference types to be carried by other types (including arrow and reference types) leads to a distinct class of behaviour. Indeed, this generalisation induces a strong notion of aliasing, in the sense that a reference name returned from a procedural call (as well as from e.g. reading references) can textually coalesce reference names in a program text. This significantly increases complexity in behaviour, hence in its logical treatment. A clean and tractable logical treatment of this phenomenon is possible on the basis of the logic studied here using ideas from the π-calculus. Details are found in our coming report [8]. On the basis of the preceding stratification, local state is also incorporated cleanly by a simple logical enrichment, reminiscent of the ν-operator in π-calculi. The full exploration of local state will be reported elsewhere. For simplicity, we have omitted polymorphism and recursive types in the present paper. Their integration is entirely straightforward following [25].

Related Work In the following we focus on directly related work, leaving more extensive comparisons to [2, 22, 23, 25].

Compositional program logics for imperative languages have been studied extensively since Hoare's seminal work. In late 1970s and early 1980s, many attempted to extend Hoare logic to higher-order languages, mostly focussing on Algol and its derivatives. Clarke [10] shows that a sound and (relatively) complete Hoare logic *cannot* exist for programming languages with a certain set of features, in particular arbitrary higher-order procedures. Clarke's argument relies on a given logical language being first-order and allowing models to have a finite universe (which makes validity in assertions recursive). As Halpern pointed out [17], a sound and complete logic may exist for higher-order programming languages if we consider other classes of models, as we do here. Olderog [39, 40], Trakhtenbrot et al. [47], German et al. [13] and Halpern [17] study Algol-like languages with procedures as parameters, obtaining various inferential relative completeness results. Unlike ours, none of these works can describe higher-order behaviour directly in assertions which we do with evaluation formulae. This restriction partly reflects the nature of their target languages, which strictly separate commands from (first-order and higher-order) expressions. Using logical languages in these works, it would be hard to capture the behaviour of examples in §2 and §6 as assertions.

Specification logic by Reynolds [44] is a program logic for Idealised Algol in the tradition of LCF, allowing higher-order programs to appear textually in assertions. For reasoning about side effects, assertions also include Hoare-triple-like formulae for command types, though their pre/post conditions only assert on first-order state, unlike our evaluation formulae. Specification logic does not have assertions on higher-order expressions in its logical language and does not allow compositional reasoning for these expressions.

Reynolds, O'Hearn and others [45] study extensions of Hoare logic in which new logical connectives are used for reasoning about low-level operations such as garbage collection in the first-order setting. A clean logical treatment of low-level features and higher-order constructs would be an interesting topic for further study.

The use of side-effect-free expressions when reasoning about assignment is a staple in compositional program logics. Freedom from side effects is however hard to maintain in the higher-order setting because of complex interplay between higher-order procedures. The clean embedding of Hoare's assignment rule in §6 suggests that the presented framework effectively refines the standard approach while retaining its virtues in the original setting. Experiment of the possible extensions in the context of an integrated verification framework such as JML [4] would be an interesting subject for further study.

Names have been used in Hoare logic since an early work by Kowaltowski [28], and are found in the work by von Oheimb [48], Leavens and Baker [31], Abadi and Leino [5] and Bierman and Parkinson [9], for treating parameter passing and return values. These works do not treat higher-order procedures and data types, which are uniformly captured in the present logic along with parameters and return values through the use of names.

None of the related works discussed above have established observational completeness in the sense of Theorem 2. The precise correspondence between contextual behaviours and logical descriptions becomes essential when we take assertions on higher-order behaviours in earnest, including in practical applications.

The origin of the assertions and judgements introduced in the present work is the logic for typed π-calculi [21, 23] where linear types lead to a compositional process logic. The known precise embeddings of high-level languages into these typed π-calculi can be used to determine the shape of name-based logics like the one presented here for the embedded languages. Once found, they can be embedded back with precision into the origi-

nating process logics. [20, 21, 23] discuss process logics and their relationship to the program logics in detail.

References

[1] C– home page. http://www.cminusminus.org.

[2] A full version of this paper. Available at: http://www.dcs.qmul.ac.uk/˜kohei/logics.

[3] Haskell home page. http://haskell.org.

[4] The Java Modeling Language (JML) home page. http://www.jmlspecs.org/.

[5] Martín Abadi and Rustan Leino. A logic for object-oriented programs. In *Verifi cation: Theory and Practice*, pages 11–41. Springer-Verlag, 2004.

[6] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. 163:409–470, 2000.

[7] K R. Apt. Ten Years of Hoare Logic: a survey. *TOPLAS*, 3:431–483, 1981.

[8] Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing for higher-order imperative functions. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, 2005.

[9] Gavin M. Bierman and Matthew J. Parkinson. Separation logic and abstractions. In *POPL'05*, 2005.

[10] E Clarke. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. In *POPL'79*, pages 129–147, 1979.

[11] Patrick Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, volume B*, pages 843–993. Elsevier, 1999.

[12] Robert W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.

[13] Steven M. German, Edmund M. Clarke, and Joseph Y. Halpern. Reasoning about procedures as parameters. In *Proc. IBM Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 206–220, 1981.

[14] Irene Greif and Albert R. Meyer. Specifying the Semantics of while Programs: A Tutorial and Critique of a Paper by Hoare and Lauer. *ACM Trans. Program. Lang. Syst.*, 3(4), 1981.

[15] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI'02*. ACM, 2002.

[16] Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.

[17] Joseph Y. Halpern. A good hoare axiom system for an algol-like language. In *Proc. 11^{th} POPL*, pages 262–271. ACM Press, 1984.

[18] Matthew Hennessy and Robin Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.

[19] Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.

[20] Kohei Honda. Sequential process logics: Soundness proofs. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, November 2003. Typescript, 50 pages.

[21] Kohei Honda. From process logic to program logic. In *Proc. ICFP'04*. ACM Press, 2004.

[22] Kohei Honda. From process logic to program logic (full version of [21]. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, November 2004. Typescript, 52 pages.

[23] Kohei Honda. Process Logic and Duality: Part (1) Sequential Processes. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, March 2004. Typescript, 234 pages.

[24] Kohei Honda and Nobuko Yoshida. Game-theoretic analysis of call-by-value computation. *TCS*, 221:393–456, 1999.

[25] Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *Proc. PPDP'04*. ACM Press, 2004.

[26] J. Martin E. Hyland and C. H. Luke Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.

[27] Thomas Kleymann. Hoare logic and auxiliary variables. Technical report, University of Edinburgh, LFCS ECS-LFCS-98-399, October 1998.

[28] Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7, 1977.

[29] Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[30] Peter Landin. A correspondence between algol 60 and church's lambda-notation. *Comm. ACM*, 8:2, 1965.

[31] Gary Leavens and Alber L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In *FM'99: World Congress on Formal Methods*. Springer, 1999.

[32] Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects.

[33] Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.

[34] Albert R. Meyer. Floyd-Hoare logic defines semantics (preface version). In *Proc. LICS'86*, 1986.

[35] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *Proc. POPL'88*, 1988.

[36] Robin Milner, Mads Tofte, and Robert W. Harper. *The Defi nition of Standard ML*. MIT Press, 1990.

[37] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[38] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[39] Ernst-Rüdiger Olderog. Sound and complete hoare-like calculi based on copy rules. *Acta Inf.*, 16:161–197, 1981.

[40] Ernst-Rüdiger Olderog. A characterization of hoare's logic for programs with pascal-like procedures. In *Proc. 15^{th} Theory of Computing*, pages 320–329, 1983.

[41] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[42] Andy M. Pitts and Ian D. B. Stark. Operational reasoning for functions with local state. In *HOOTS'98*, CUP, pages 227–273, 1998.

[43] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, DAIMI, Aarhus University, 1981.

[44] John C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.

[45] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. LICS'02*, 2002.

[46] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

[47] Boris A. Trakhtenbrot, Joseph Y. Halpern, and Albert R. Meyer. From denotational to operational and axiomatic semantics for algol-like languages. In *Proc. CMU Workshop on Logic of Programs*, pages 474–500, 1984.

[48] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13), 2001.